

Heap Reference Analysis for Functional Programs

Amey Karkare*, Amitabha Sanyal, and Uday Khedker

Department of CSE, IIT Bombay
Mumbai, India
{karkare,as,uday}@cse.iitb.ac.in

Abstract. Current garbage collectors leave a lot of garbage uncollected because they conservatively approximate liveness by reachability from program variables. In this paper, we describe a sequence of static analyses that takes as input a program written in a first-order, eager functional programming language, and finds at each program point the references to objects that are guaranteed not to be used in the future. Such references are made null by a transformation pass. If this makes the object unreachable, it can be collected by the garbage collector. This causes more garbage to be collected, resulting in fewer collections. Additionally, for those garbage collectors which scavenge live objects, it makes each collection faster.

The interesting aspects of our method are both in the identification of the analyses required to solve the problem and the way they are carried out. We identify three different analyses — *liveness*, *sharing* and *accessibility*. In liveness and sharing analyses, the function definitions are analyzed independently of the calling context. This is achieved by using a variable to represent the unknown context of the function being analyzed and setting up constraints expressing the effect of the function with respect to the variable. The solution of the constraints is a summary of the function that is parameterized with respect to a calling context and is used to analyze function calls. As a result we achieve context sensitivity at call sites without analyzing the function multiple number of times.

1 Introduction

An object is dead at an execution instant if it is not used in future. Ideally, garbage collectors should reclaim all objects that are dead at the time of garbage collection. However, even state of the art garbage collectors are not able to distinguish between reachable objects that are live and reachable objects that are dead. Therefore they conservatively approximate the liveness of an object by its reachability from a set of locations called the *root set* (stack locations and registers containing program variables). As a consequence, many dead objects

* Supported by Infosys Technologies Limited, Bangalore, under Infosys Fellowship Award.

are left uncollected. This has been confirmed by empirical studies for Haskell [1], Scheme [2] and Java [3–5].

In this paper, we consider a first order functional language without imperative features and propose a method to release dead objects so that they can be collected by the garbage collector. This is done by detecting unused references to objects and setting them to null. If all references to the object are nullified, then the dead objects may become unreachable and may be claimed by garbage collector. We propose three analyses to obtain the information required for nullification: *liveness* analysis, which computes live references at each program point (i.e. the references used by the program beyond the program point), *sharing* analysis, which computes alternate ways to access live references and *accessibility* analysis which ensures that the references used by the nullification statement itself exist and do not cause a dereferencing exception. An earlier paper [6] outlined the basic method and provided details of the liveness analysis. This paper brings the theoretical aspects of the method to completion.

As our analyses are interprocedural in scope, the effect of function calls on the heap must be modeled precisely. Most program analyses are either not scalable because they analyze the same function more than once or imprecise because they make overly safe worst-case assumptions about the effect of a function on the heap. For a better balance between scalability and precision, one can compute context independent summaries of the effect of functions on the heap and then use this summary at particular calling context of the function [7–9]. We do this by using a variable to represent an unknown context of the function being analyzed and setting up constraints expressing the effect of the function with respect to the variable. The set of constraints is viewed as a set of CFGs and the solution of these constraints is a set of finite state machines approximating the languages defined by the CFGs. The solution, which is a summary of the function parameterized with respect to a calling context, is used to analyze function calls.

The main contributions of the paper are as follows. We identify the analysis required to find nullable references at each program point. As part of the analyses, we show how context independent summaries of functions can be obtained by setting up a set of constraints and solving them by viewing them as a CFG. Finally we show how the result can be used for safe insertion of nullifying statements in the program.

1.1 Motivation

Figure 1(a) shows an example program. The label π of an expression e denotes the program point just before the evaluation of e . The heap memory can be viewed as a (possibly unconnected) directed acyclic graph called *memory graph*¹ during any instant in the execution of the program. The elements of root set are the entry points for the memory graph. The nodes in the memory graph are the **cons** cells allocated in the heap. There are three kind of edges in the memory graph: (1) Entry edges from an element of the root set to a heap node,

¹ Since the language under consideration (Sec. 2) does not have any imperative features, the memory graph can not have cycles.

| | | |
|---------|---|--|
| $p ::=$ | $d_1 \dots d_n e_{\text{pgm}}$ | — <i>program</i> |
| $d ::=$ | $(\text{define } (f \ v_1 \dots v_n) \ e_1)$ | — <i>function definition</i> |
| $e ::=$ | | — <i>expression</i> |
| | κ | — <i>constant</i> |
| | $ \ v$ | — <i>variable</i> |
| | $ \ \text{nil}$ | $ \ (\text{cons } e_1 \ e_2)$ — <i>constructors</i> |
| | $ \ (\text{car } e_1)$ | $ \ (\text{cdr } e_1)$ — <i>selectors</i> |
| | $ \ (\text{pair? } e_1)$ | $ \ (\text{null? } e_1)$ — <i>testers</i> |
| | $ \ (+ \ e_1 \ e_2)$ | — <i>generic primitive</i> |
| | $ \ (\text{if } e_1 \ e_2 \ e_3)$ | — <i>conditional</i> |
| | $ \ (\text{let } v_1 \leftarrow e_2 \ \text{in } e_3)$ | — <i>let binding</i> |
| | $ \ (f \ e_1 \dots e_n)$ | — <i>function application</i> |

Fig. 2. The syntax of our language

1.2 Organization

Section 2 describes the language used to explain our analysis along with the basic concepts and notations. Liveness analysis is described in Section 3. Section 4 explains the analysis to compute sharing between root variables. Section 5 describes availability analysis. Section 6 describes the actual process of null insertion. The related work is given in Section 7. We conclude in Section 8 and give the direction for future research.

2 Concepts and Notations

The syntax of our language is shown in Fig. 2. The language has call-by-value semantics. The argument expressions are evaluated from left to right. We assume that variables in the program are renamed so that the same name is not defined in multiple scopes. The body of the program is the expression denoted by e_{pgm} . We write $\pi : e$ to associate π with the program point just before the expression e .

An edge emanating from a **car** field is labeled **0** while an edge emanating from a **cdr** field is labeled **1**. Entry edges do not have any label. There are two kinds of traversals associated with an edge: A *forward* traversal is in the direction of the edge, and a *backward* traversal is in the opposite direction of the edge. For an edge with label $l (l \in \{0, 1\})$, a forward traversal over the edge is denoted by l , while \bar{l} denotes a backward traversal over the edge.

Given a node in a memory graph, a *path* is a sequence of labels representing a traversal over connected edges starting at the node. In general, a path involves both forward as well as backward traversals over edges. A *forward* path involves only forward traversals over edges, and a *backward* path involves only backward traversals over edges. Note that starting from a **cons** cell there can be multiple possible edge traversals labeled $\bar{0}$ or $\bar{1}$, but at most one traversal labeled **0** or **1**. In general, all forward traversals from a node have unique labels while multiple backward traversals may share the same label. A *bipath* consists of a (possibly empty) forward path followed by a (possibly empty) backward path. Note that

forward and backward paths are special cases of bipath. Only bipaths are important to us because liveness, sharing and accessibility can all be described using bipaths. We use Greek letters (α, β, \dots) to denote paths. The concatenation of two path segments α and β is denoted by $\alpha\beta$. The reverse of a path α , denoted $\bar{\alpha}$, is the path which traverses the edges of α in the opposite order and opposite direction. We have: $\bar{\bar{\alpha}} = \alpha$, and $\overline{\alpha_1\alpha_2} = \bar{\alpha_2}\bar{\alpha_1}$. The concatenation ($\sigma_1 \cdot \sigma_2$) of a set of paths σ_1 with σ_2 is defined as a set containing concatenation of each element in σ_1 with each element in σ_2 .

A path can be simplified by repeatedly removing consecutive occurrences of backward and forward traversal of the same edge (in general, removing occurrences of $\bar{\alpha}\alpha$). The reduction does not change the semantics of the path in that the node reached by the path remains the same even after simplification. Further, since we are interested in bipaths only, paths containing $\bar{1}0$ or $0\bar{1}$ can be ignored. This gives us the following rules of reduction:

$$\begin{array}{lll} \alpha_1\bar{0}0\alpha_2 \rightarrow \alpha_1\alpha_2 & \alpha_1\bar{1}1\alpha_2 \rightarrow \alpha_1\alpha_2 & \alpha_1\perp\alpha_2 \rightarrow \perp \\ \alpha_10\bar{1}\alpha_2 \rightarrow \perp & \alpha_11\bar{0}\alpha_2 \rightarrow \perp & \end{array} \quad (1)$$

$\alpha \xrightarrow{k} \alpha'$ denotes the reduction of α to α' in k steps, and $\xrightarrow{*}$ denotes the reflexive and transitive closure of \rightarrow . A path which can not be reduced further using above rules is said to be in *canonical* form. Note that a path in canonical form is either a bipath or \perp .

Very often we shall be interested in paths that start from a heap cell pointed directly by the root set. We call such paths as *access paths*. Let $\text{Loc}[e]$ denote the stack location which holds the value of e^2 and $\text{Cell}[e]$ denote the heap node pointed to by $\text{Loc}[e]$. We use $e.\alpha$ to denote access path which starts in the heap at $\text{Cell}[e]$ and traverse the path α . If σ denotes a set of paths, then $e.\sigma$ is the set of access paths rooted at $\text{Cell}[e]$ corresponding to σ . i.e. $e.\sigma = \{e.\alpha \mid \alpha \in \sigma\}$. We use access paths to refer to links in the memory graph. The link referred to by an access path is the last edge in a traversal using the access path.

The syntax of the meta-language used to describe our analysis is very similar to the language being analyzed. To distinguish between them, the keywords in the meta-language are written in all capitals (**LET**, **IN**, **IF** etc.).

3 Liveness Analysis

A link in a memory graph is live at a program point π if some expression dereferences it beyond π . An access path is live if the link denoted by it is live. Note that an access path can be live in two ways: either it is used directly to access the last link, or it shares the live link with some other access path using which the link is made live. Liveness analysis discovers access paths through which the live link is directly accessed.

² For a root variable r , $\text{Loc}[r]$ is same as r . For any other expression e , $\text{Loc}[e]$ can be thought of as the temporary that holds the value of e .

The *liveness environment* at π , denoted \mathcal{L}_π , describes all the live access paths at π . It is a function from root variables to sets of paths. The result of liveness analysis is the annotation of each program point with its liveness environment. The liveness of an access path before an expression e depends upon its use inside e itself and in the rest of the program through the result of e . Therefore we define a transfer function denoted \mathcal{LE} to compute the liveness of access paths before an expression, given the liveness of result after the expression. As expressions may contain applications of primitive operations and functions, we also need to propagate liveness across these applications. This is done through the summarizing functions \mathcal{LP} and \mathcal{LF} . While \mathcal{LP} is given directly based on the semantics of the primitive, \mathcal{LF} is inferred from the body of a function.

3.1 Liveness Transfer Function (\mathcal{LE})

For an expression $\pi : e$, a set of paths σ specifying the liveness of the result of evaluating e and the liveness environment \mathcal{L} after e , $\mathcal{LE}(e, \sigma, \mathcal{L})$ computes the liveness environment at π . The liveness environment associated with the exit of any function is empty liveness environment \mathcal{L}^\emptyset defined as $\forall x \mathcal{L}^\emptyset(x) = \emptyset$. The liveness associated with the result of the program expression e_{pgm} is $\sigma_{\text{pgm}} = \{\mathbf{0}, \mathbf{1}\}^*$, i.e. the entire result of the program is needed. For any other function f , the liveness associated with the result is:

$$\sigma_{\text{exit}_f} = \bigcup_{\text{all calls } (f \ e_1 \dots e_n)} \{\sigma \mid \sigma \text{ is the liveness of the result of } (f \ e_1 \dots e_n) \text{ after the call}\}$$

The computation of \mathcal{LE} is given in Fig. 3.³ In the expression (**if** $e_1 \ e_2 \ e_3$), the σ for e_1 is $\{\epsilon\}$ because the value of e_1 is used to decide the branch, for which only $\text{Cell}[e_1]$ is used (4). For a **let**, the liveness of v_1 from e_2 and beyond is transferred to e_1 (5). The liveness environment before a primitive application is computed by using \mathcal{LP} to transfer the liveness from the result of the application to each of its arguments (6). Similarly, applications of user defined functions use \mathcal{LF} (7).

As the result of liveness analysis is the annotation of every program point with its liveness environment, during computation of $\mathcal{LE}(e, \sigma, \mathcal{L})$, the program point before e is annotated with the computed liveness environment as a side effect. We do not show this explicitly to avoid clutter.

3.2 Summarizing Functions (\mathcal{LP} and \mathcal{LF})

If σ describes the set of paths specifying the liveness of the result of $(P \ e_1 \dots e_n)$ after the call, then $\mathcal{LP}_P^i(\sigma)$ gives the set of access paths specifying the liveness of e_i at the program point after e_i . The summarizing functions for the primitives in our language, **car**, **cdr**, **cons**, **null?**, **pair?** and **+**, are shown below. The 0-ary constructor **nil** does not accept any argument and is ignored.

$$\begin{aligned} \mathcal{LP}_{\text{car}}^1(\sigma) &= \{\epsilon\} \cup \{\mathbf{0}\} \cdot \sigma & \mathcal{LP}_{\text{cdr}}^1(\sigma) &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \sigma \\ \mathcal{LP}_{\text{cons}}^1(\sigma) &= \{\mathbf{0}\} \cdot \sigma & \mathcal{LP}_{\text{cons}}^2(\sigma) &= \{\mathbf{1}\} \cdot \sigma \\ \mathcal{LP}_{\text{null?}}^1(\sigma) &= \{\epsilon\}, & \mathcal{LP}_{\text{pair?}}^1(\sigma) &= \{\epsilon\}, & \mathcal{LP}_+^1(\sigma) &= \{\epsilon\}, & \mathcal{LP}_+^2(\sigma) &= \{\epsilon\} \end{aligned} \quad (8)$$

³ update is a helper function to compute change in environments:

$\text{update}(\text{OldEnv}, Y, \text{NewVal})(X) = (\text{IF } (X == Y) \text{ NewVal } (\text{OldEnv } X))$

$$\mathcal{LE}(\kappa, \sigma, \mathcal{L}) = \mathcal{L} \quad (2)$$

$$\mathcal{LE}(v, \sigma, \mathcal{L}) = \text{update}(\mathcal{L}, v, \mathcal{L}(v) \cup \sigma) \quad (3)$$

$$\begin{aligned} \mathcal{LE}((\text{if } e_1 \ e_2 \ e_3), \sigma, \mathcal{L}) = & (\text{LET } \mathcal{L}' \leftarrow \mathcal{LE}(e_3, \sigma, \mathcal{L}) \text{ IN} \\ & (\text{LET } \mathcal{L}'' \leftarrow \mathcal{LE}(e_2, \sigma, \mathcal{L}) \text{ IN} \\ & \mathcal{LE}(e_1, \{\epsilon\}, \mathcal{L}' \cup \mathcal{L}'')))) \end{aligned} \quad (4)$$

$$\begin{aligned} \mathcal{LE}((\text{let } v_1 \leftarrow e_1 \text{ in } e_2), \sigma, \mathcal{L}) = & (\text{LET } \mathcal{L}' \leftarrow \mathcal{LE}(e_2, \sigma, \mathcal{L}) \text{ IN} \\ & \mathcal{LE}(e_1, \mathcal{L}'(v_1), \text{update}(\mathcal{L}', v_1, \emptyset))) \end{aligned} \quad (5)$$

$$\begin{aligned} \mathcal{LE}((P \ e_1 \ \dots \ e_n), \sigma, \mathcal{L}) = & (\text{LET } \mathcal{L}_1 \leftarrow \mathcal{LE}(e_n, \mathcal{LP}_P^n(\sigma), \mathcal{L}) \text{ IN} \\ & P \text{ is a primitive } (\text{LET } \mathcal{L}_2 \leftarrow \mathcal{LE}(e_{n-1}, \mathcal{LP}_P^{n-1}(\sigma), \mathcal{L}_1) \text{ IN} \\ & \dots \\ & (\text{LET } \mathcal{L}_{n-1} \leftarrow \mathcal{LE}(e_2, \mathcal{LP}_P^2(\sigma), \mathcal{L}_{n-2}) \text{ IN} \\ & \mathcal{LE}(e_1, \mathcal{LP}_P^1(\sigma), \mathcal{L}_{n-1}))) \dots) \end{aligned} \quad (6)$$

$$\begin{aligned} \mathcal{LE}((f \ e_1 \ \dots \ e_n), \sigma, \mathcal{L}) = & (\text{LET } \mathcal{L}_1 \leftarrow \mathcal{LE}(e_n, \mathcal{LF}_f^n(\sigma), \mathcal{L}) \text{ IN} \\ & f \text{ is a user defined function } (\text{LET } \mathcal{L}_2 \leftarrow \mathcal{LE}(e_{n-1}, \mathcal{LF}_f^{n-1}(\sigma), \mathcal{L}_1) \text{ IN} \\ & \dots \\ & (\text{LET } \mathcal{L}_{n-1} \leftarrow \mathcal{LE}(e_2, \mathcal{LF}_f^2(\sigma), \mathcal{L}_{n-2}) \text{ IN} \\ & \mathcal{LE}(e_1, \mathcal{LF}_f^1(\sigma), \mathcal{L}_{n-1}))) \dots) \end{aligned} \quad (7)$$

Fig. 3. Computing \mathcal{LE}

$\mathcal{LP}_{\mathbf{car}}^1(\sigma)$ includes $\{\mathbf{0}\} \cdot \sigma$ because the link described by a path labeled α from $\text{Cell}[(\mathbf{car} \ e)]$ can also be described by the path labeled $\mathbf{0}\alpha$ from $\text{Cell}[e]$. Also, as the cell corresponding to e is used to find the value of \mathbf{car} , we need to add ϵ to the live paths of e . Reasoning about $(\mathbf{cdr} \ e)$ is similar. For similar reasons, a path α describing the liveness of \mathbf{cons} translates to an $\mathbf{0}\alpha$ for its first argument, and $\mathbf{1}\alpha$ for its second argument. Further, as \mathbf{cons} does not read its arguments, the access paths of the arguments do not contain ϵ . The remaining primitives read only the value of the arguments, therefore the set of live path of the arguments is $\{\epsilon\}$.

\mathcal{LF} plays the same role as \mathcal{LP} for user defined functions. Given a function defined as $(\mathbf{define} \ (f \ v_1 \ \dots \ v_n) \ e)$ and a σ specifying the set of paths specifying the liveness of the result, \mathcal{LF} is computed as follows:

$$\mathcal{LF}_f^i(\sigma) = \mathcal{LE}(e, \sigma, \emptyset)(v_i), \ 1 \leq i \leq n \quad (9)$$

Example 1. To compute the transfer functions for **append**, we compute $\mathcal{LE}(e, \sigma, \emptyset)$ in terms of a variable σ . Here e is the body of **append**. Figure 4 shows the values at various program points in **append**. From the liveness information of the parameters **lst1** and **lst2**, we get:

$$\begin{aligned} \mathcal{LF}_{\mathbf{append}}^1(\sigma) &= \{\epsilon\} \cup \{\mathbf{00}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{LF}_{\mathbf{append}}^1(\{\mathbf{1}\} \cdot \sigma) \\ \mathcal{LF}_{\mathbf{append}}^2(\sigma) &= \sigma \cup \mathcal{LF}_{\mathbf{append}}^2(\{\mathbf{1}\} \cdot \sigma) \end{aligned}$$

□

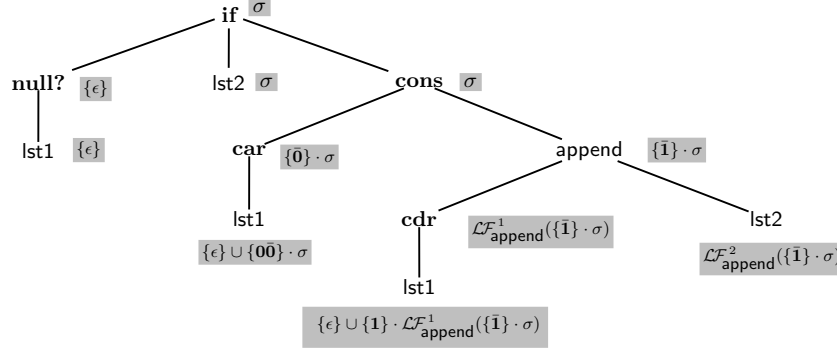


Fig. 4. Transformation of access paths for body of `append`

3.3 Solving Liveness Equations

We now describe briefly the steps to solve the liveness equations. The reference [6] and the Appendix A both contain a detailed example illustrating these steps. Further, the equations resulting out of the sharing analysis are also solved in a similar manner.

In general, the equations defining the functions \mathcal{LF} will be recursive. To solve such equations we start by guessing that the solution for $\mathcal{LF}_f^i(\sigma)$ will be of the form: $\mathcal{I}_f^i \cup \mathcal{D}_f^i \cdot \sigma$, where \mathcal{I}_f^i and \mathcal{D}_f^i are sets of strings over the alphabet $\{0, 1, \bar{0}, \bar{1}\}$. Then,

1. We substitute the guessed form of \mathcal{LF}_f^i in the equations and equate the σ -dependent and σ -independent parts of LHS and RHS of each equation. This gives us equations for \mathcal{I}_f^i and \mathcal{D}_f^i which are independent of σ .
2. We interpret the equations as rules of a context free grammar (CFG) with \mathcal{I}_f^i and \mathcal{D}_f^i as non-terminals. The set of terminal symbols of the CFG is $\{0, 1, \bar{0}, \bar{1}\}$.
3. We add more rules to represent the liveness at different program points in terms of the above non-terminals.
4. We approximate the CFG by a set of non deterministic finite automata (NFA) and simplify the NFAs so that the paths in canonical form are accepted. The algorithm describing this step and its proof of correctness is given in Appendix A.2. The algorithm is a revised version of that given in our earlier work [6].

4 Sharing Analysis

Given a memory graph, expressions e_1 and e_2 are involved in sharing if there are forward paths from $\text{Cell}[e_1]$ and $\text{Cell}[e_2]$ to a common heap cell. In particular we are interested in the sharing of the root variables. Let h be a heap cell shared by root variables x and y . Let the forward access path $x.\alpha$ describe the path

| | | | |
|--------------------|--|---|--|
| Expression: | $(\text{let } y_1 \leftarrow (\text{car } x_1) \text{ in } \pi_1 : \dots)$ | $(\text{define } (f \ v_1 \ v_2) \ \pi_2 : \dots) \dots$ $(f \ (\text{car } x_2) \ x_2)$ | $(\text{let } y_3 \leftarrow (\text{cons } x_3 \ x_3) \text{ in } \pi_3 : \dots)$ |
| Sharing: | $\mathbf{0} \in \mathcal{S}_{\pi_1}(x_1, y_1)$ | $\bar{\mathbf{0}} \in \mathcal{S}_{\pi_2}(v_1, v_2)$ | $\mathbf{0}\bar{\mathbf{1}}, \mathbf{1}\bar{\mathbf{0}} \in \mathcal{S}_{\pi_3}(y_3, y_3); \bar{\mathbf{0}}, \bar{\mathbf{1}} \in \mathcal{S}_{\pi_3}(x_3, y_3)$ |

Fig. 5. Examples of sharing

from x to h and the forward access path $y.\beta$ describe the path from y to h . Then, sharing between x and y can be seen as a bipath labeled $\alpha\bar{\beta}$ from $\text{Cell}[x]$ to $\text{Cell}[y]$ in the memory graph. Fig. 5 shows some ways in which sharing can arise.

The *sharing environment* at π , denoted \mathcal{S}_π , describes the sharing between root variables in any memory graph that can arise at π . The sharing environment is a function from pairs of root variables to sets of bipaths. The result of sharing analysis is to annotate each program point with an approximation of its sharing environment.

Since variables take their values from evaluation of expressions (through **let** or argument bindings), it is convenient to define a function denoted \mathcal{S} , which computes the sharing between variables and expressions. Further, we also need to propagate sharing environments across applications of primitive operations and user defined functions. This is done by using the summarizing functions \mathcal{SP} and \mathcal{SF} . For a primitive P , \mathcal{SP}_P^i denotes the sharing between the i^{th} argument and the result of P . \mathcal{SF}_f^i is interpreted in a similar manner. Additionally the function \mathcal{SS} computes the sharing of an expression with itself.

4.1 Sharing Transfer Function (\mathcal{S})

The transfer function $\mathcal{S}(x, e, \mathcal{S})$ computes the extent of sharing between the root variable x and the result obtained by evaluating e .⁴ In our language, the sharing between root variables can only be affected either at the **let**-binding or at the entry of a function. The computation of \mathcal{S} begins at function definitions. The sharing environment before the program expression e_{pgm} is the empty sharing environment \mathcal{S}^0 defined as $\forall x, y \ \mathcal{S}^0(x, x) = \{\epsilon\}, \mathcal{S}^0(x, y) = \emptyset$. For any other function f , defined as **(define** $(f \ v_1 \dots v_n) \ e)$, the initial sharing environment $\mathcal{S}_{\text{entry}_f}$ is as shown in Fig. 6.

The computation of $\mathcal{S}(x, e, \mathcal{S})$ is given in Fig. 7. Equations (10) and (11) are self-explanatory. In an **if** expression, sharing can be due to execution of either branch. The sharing between x and e_1 is computed to propagate the sharing environment inside e_1 ; it does not affect the sharing between x and the **if** expression. For a **let** expression, sharing environment \mathcal{S}' at e_2 captures the sharing between x and v_1 (13). Finally, the sharing between x and the result of application of a primitive P is obtained by composing the sharing between x and e_i with the sharing between the e_i and the result (14). User defined functions

⁴ The function can easily be extended to a set of variables so that only a single pass over the expression is required.

$$\begin{aligned}
\mathcal{S}_{entry_f}(v_i, v_i) &= \{\epsilon\} \cup \bigcup_{\pi: (f \ e_1 \dots e_n)} \mathcal{SS}(e_i, \mathcal{S}_\pi) \\
\mathcal{S}_{entry_f}(v_i, v_j) &= \bigcup_{\pi: (f \ e_1 \dots e_n)} \mathcal{EES}(e_i, e_j, \mathcal{S}_\pi, \mathbf{SVars}(\pi)) \\
&\text{where } 1 \leq i, j \leq n, \ i \neq j, \ \mathbf{SVars}(\pi) = \text{set of root variables in scope at } \pi \\
\mathcal{EES}(e, e', \mathcal{S}, \text{Vars}) &= \{\bar{\alpha}\beta \mid \alpha \in \mathcal{SE}(x, e, \mathcal{S}), \beta \in \mathcal{SE}(x, e', \mathcal{S}), x \in \text{Vars}\}
\end{aligned}$$

Fig. 6. Sharing at the entry of a function

$$\mathcal{SE}(x, \kappa, \mathcal{S}) = \emptyset \quad (10)$$

$$\mathcal{SE}(x, v, \mathcal{S}) = \mathcal{S}(x, v) \quad (11)$$

$$\begin{aligned}
\mathcal{SE}(x, (\text{if } e_1 \ e_2 \ e_3), \mathcal{S}) &= (\mathbf{LET} \ \mathcal{S}' \leftarrow \mathcal{SE}(x, e_1, \mathcal{S}) \ \mathbf{IN} \quad \{\mathcal{S}' \text{ is ignored}\} \\
&\quad \mathcal{SE}(x, e_2, \mathcal{S}) \cup \mathcal{SE}(x, e_3, \mathcal{S})) \quad (12)
\end{aligned}$$

$$\begin{aligned}
\mathcal{SE}(x, (\text{let } v_1 \leftarrow e_1 \ \text{in } e_2), \mathcal{S}) &= (\mathbf{LET} \ \mathcal{S}' \leftarrow \text{update}(\mathcal{S}, (v_1, v_1), \{\epsilon\} \cup \mathcal{SS}(e_1, \mathcal{S})) \ \mathbf{IN} \\
&\quad (\mathbf{LET} \ \mathcal{S}'' \leftarrow \text{update}(\mathcal{S}', (x, v_1), \mathcal{SE}(x, e_1, \mathcal{S})) \ \mathbf{IN} \\
&\quad \mathcal{SE}(x, e_2, \mathcal{S}'')) \quad (13)
\end{aligned}$$

$$\begin{aligned}
\mathcal{SE}(x, (P \ e_1 \dots e_n), \mathcal{S}) &= \bigcup_{\substack{P \text{ is a primitive} \\ 1 \leq i \leq n}} \mathcal{SE}(x, e_i, \mathcal{S}) \cdot \mathcal{SP}_P^i \quad (14)
\end{aligned}$$

$$\begin{aligned}
\mathcal{SE}(x, (f \ e_1 \dots e_n), \mathcal{S}) &= \bigcup_{\substack{f \text{ is a user defined function} \\ 1 \leq i \leq n}} \mathcal{SE}(x, e_i, \mathcal{S}) \cdot \mathcal{SF}_f^i \quad (15)
\end{aligned}$$

Fig. 7. Computing \mathcal{SE}

(15) are treated similarly. Note that only **let** expression modifies the sharing environment. During computation of $\mathcal{SE}(x, e, \mathcal{S})$ the program point before e is annotated with \mathcal{S} . However, as in \mathcal{LE} , we do not show this explicitly.

4.2 Summarizing Functions (\mathcal{SP} and \mathcal{SF})

\mathcal{SP} specifies the extent of sharing between the formal arguments of a primitive and its return value. The sharing between i^{th} argument and the result is denoted by \mathcal{SP}_P^i . For a primitive application $(P \ e_1 \dots e_n)$:

$$\alpha\bar{\beta} \in \mathcal{SP}_P^i \Rightarrow \text{there is a bipath } \alpha\bar{\beta} \text{ from } \text{Loc}[e_i] \text{ to } \text{Loc}[(P \ e_1 \dots e_n)]$$

The functions \mathcal{SP}_P^i , of a primitive are computed from its semantics:

$$\begin{aligned}
\mathcal{SP}_{\text{car}}^1 &= \{0\} & \mathcal{SP}_{\text{cdr}}^1 &= \{1\} & \mathcal{SP}_{\text{cons}}^1 &= \{\bar{0}\} & \mathcal{SP}_{\text{cons}}^2 &= \{\bar{1}\} \\
\mathcal{SP}_{\text{null?}}^1 &= \emptyset & \mathcal{SP}_{\text{pair?}}^1 &= \emptyset & \mathcal{SP}_{+}^1 &= \emptyset & \mathcal{SP}_{+}^2 &= \emptyset
\end{aligned} \quad (16)$$

\mathcal{SF} specifies the extent of sharing between the formal arguments of a function and its return value. The sharing between i^{th} argument and the result is denoted by \mathcal{SF}_f^i . For a function defined as (**define** $(f \ v_1 \dots v_n) \ e$), \mathcal{SF}_f^i is computed as

$$\mathcal{SS}(\kappa, \mathcal{S}) = \{\epsilon\} \quad (18)$$

$$\mathcal{SS}(v, \mathcal{S}) = \mathcal{S}(v, v) \quad (19)$$

$$\mathcal{SS}((\text{if } e_1 \ e_2 \ e_3), \mathcal{S}) = \mathcal{SS}(e_2, \mathcal{S}) \cup \mathcal{SS}(e_3, \mathcal{S}) \quad (20)$$

$$\begin{aligned} \mathcal{SS}((\text{let } v_1 \leftarrow \pi_1 : e_1 \text{ in } e_2), \mathcal{S}) &= (\text{LET } \mathcal{S}_0 \leftarrow \text{update}(\mathcal{S}, (v_1, v_1), \mathcal{SS}(e_1, \mathcal{S})) \text{ IN} \\ \text{SVars}(\pi_1) = \{x_1, \dots, x_n\} &\quad (\text{LET } \mathcal{S}_1 \leftarrow \text{update}(\mathcal{S}_0, (x_1, v_1), \mathcal{E}(x_1, e_1, \mathcal{S})) \text{ IN} \\ &\dots \\ &\quad (\text{LET } \mathcal{S}_n \leftarrow \text{update}(\mathcal{S}_{n-1}, (x_n, v_1), \mathcal{E}(x_n, e_1, \mathcal{S})) \text{ IN} \\ &\quad \mathcal{SS}(e_2, \mathcal{S}_n)) \dots) \end{aligned} \quad (21)$$

$$\begin{aligned} \mathcal{SS}(\pi : (P \ e_1 \ \dots \ e_n), \mathcal{S}) &= \bigcup_{\substack{1 \leq i, j \leq n \\ i \neq j}} \overline{\mathcal{SP}_P^i} \cdot \left(\bigcup_{\pi} \mathcal{ES}(e_i, e_j, \mathcal{S}, \text{SVars}(\pi)) \right) \cdot \mathcal{SP}_P^j \\ &\quad \cup \bigcup_{1 \leq i \leq n} \overline{\mathcal{SP}_P^i} \cdot \mathcal{SS}(e_i, \mathcal{S}) \cdot \mathcal{SP}_P^i \end{aligned} \quad (22)$$

$$\begin{aligned} \mathcal{SS}(\pi : (f \ e_1 \ \dots \ e_n), \mathcal{S}) &= \bigcup_{\substack{1 \leq i, j \leq n \\ i \neq j}} \overline{\mathcal{SF}_f^i} \cdot \left(\bigcup_{\pi} \mathcal{ES}(e_i, e_j, \mathcal{S}, \text{SVars}(\pi)) \right) \cdot \mathcal{SF}_f^j \\ &\quad \cup \bigcup_{1 \leq i \leq n} \overline{\mathcal{SF}_f^i} \cdot \mathcal{SS}(e_i, \mathcal{S}) \cdot \mathcal{SF}_f^i \end{aligned} \quad (23)$$

Fig. 8. Computing \mathcal{SS}

follows:

$$\mathcal{SF}_f^i = \mathcal{E}(v_i, e, \mathcal{S}^\emptyset), \ 1 \leq i \leq n \quad (17)$$

4.3 Sharing with Self (\mathcal{SS})

Because of the sharing in the subexpressions, the result of an expression may share a **cons** cell along two different paths. We call it self sharing, and use the function \mathcal{SS} to capture it. The computation of \mathcal{SS} is shown in Fig. 8.

4.4 Computing Aliases of Access Paths

We say that two access paths are *aliased* at a program point if they share the same **cons** cell in the heap at that point. We distinguish between two kinds of aliases: two access paths are *link*-aliases if they share the last edge in the path, otherwise they are *node*-aliases.

The result of sharing analysis can be used to compute all aliases of a given access path at a given point. Let π be a program point, and let \mathcal{S}_π be the sharing environment at π . Further, let $x.\alpha$ be an access path under consideration, where α is a forward path. To find out the aliases of $x.\alpha$ rooted at y , we proceed as follows. Consider the set $\mathcal{S}_\pi(y, x)$ which contains the bipaths from $\text{Cell}[y]$ to $\text{Cell}[x]$. For $\beta \in \mathcal{S}_\pi(y, x)$, if $\beta\alpha$ reduces to a forward path then $y.\beta\alpha$ is a forward access path which reaches the same **cons** cell as that reached by $x.\alpha$ implying that $y.\beta\alpha$ is an alias of $x.\alpha$. Because we do not have the bipaths in

\mathcal{S}_pi explicitly listed, we have to compute CFGs describing the bipaths. This is same as described for liveness (App. A, [6]). We also compute the trivial CFG describing the string α . The concatenation of CFG describing $\mathcal{S}_\pi(y, x)$ with CFG describing the string α gives a CFG, which after conversion to NFA and simplification gives the regular grammar describing the aliases of $x.\alpha$ rooted at y .

The link alias of a root variable x is x itself. To get the link-aliases of $x.\alpha\mathbf{0}$, we compute aliases of $x.\alpha$ as described above, and extend it by $\mathbf{0}$. Similarly we can compute link-aliases for $x.\alpha\mathbf{1}$.

5 Accessibility Analysis

To nullify a link l at a program point π , we have to traverse an access path from some root variable, say v , to the source cell of l . However, it is possible that some **cons** cell c in the access path from v to l is created along one execution path to π but not along another. Since the nullification of l at π requires the cell c to be dereferenced, a run time exception may occur if the execution path taken is the one along which c is not created. To avoid this, we need to make sure that the access path used for nullification is such that all the intermediate cells in it are definitely created.

Example 2. Consider the following program fragment:

(let $x \leftarrow$ (if ($y < 5$) (cons 2 z) nil) in π :(if ($y \geq 5$) $\pi_1:w$ $\pi_2:(\text{cdr } x)$))

Observe that the program does not raise a dereferencing exception. Assume that the link $x.0$ is not live at π . This information is not sufficient to nullify $x.0$ safely at π because it does not guarantee that variable x points to a **cons** cell at π . Similarly, knowing that $x.0$ is not live at π_1 or π_2 does not enable us to nullify $x.0$ at those points. However, since $(\text{cdr } x)$ dereferences x , we can infer that x can be dereferenced at π_2 . Thus, we can safely nullify $x.0$ at π_2 . \square

Assuming that the program cannot generate a dereferencing exception, it is possible to infer the set of access paths that can be dereferenced without causing exception. We call such paths *accessible*. There are two ways in which the set of accessible paths can be inferred at π . We can discover access paths in which all the **cons** cells are either created or dereferenced along all program paths from the program entry to π . We call these paths as *available* paths at π . Secondly, we can discover access paths in which all the **cons** cells are dereferenced along all program paths from π to the program exit. We call these paths as *anticipable*.

In this paper we describe availability analysis only. The result of availability analysis is the *availability environment*, denoted \mathcal{A} , corresponding to each program point. It is a function from root variables to the corresponding available access paths. We next describe how the availability environment is computed.

5.1 Availability Transfer Function (\mathcal{AE})

In general, an expression has to dereference the structures corresponding to its subexpressions. Therefore, for execution to proceed normally, these structures

must exist. We call this requirement as the *demand* on a subexpression. We use a set of paths to describe the demand. The demand from the enclosing expression is modified by an expression and passed to its subexpressions.

Example 3. **car** and **cdr** require that their arguments are non null. Thus, for expression $(\mathbf{car} \ \pi_1 : (\mathbf{cdr} \ \pi_2 : x))$, the demand at π_1 is $\{\epsilon\}$ due to the **car** application. The demand at π_2 is $\{\epsilon, \mathbf{1}\}$, where ϵ is due to **cdr**, and $\mathbf{1}$ is because of the demand of **car** on $(\mathbf{cdr} \ x)$ which is modified and passed to x . \square

The way availability information is generated and propagated is as follows. Consider an expression $(\mathbf{car} \ (\mathbf{cdr} \ x))$. Assume that the availability environment before this expression indicates that no access path rooted at x is available. When we reach the subexpression x , the chain of selectors $(\mathbf{car} \ (\mathbf{cdr} \ \dots$ generates the demand $\{\epsilon, \mathbf{1}\}$ on x . We thus update the availability environment of x to include $x.\{\epsilon, \mathbf{1}\}$. This availability is propagated upwards and used to conclude that the availability of $(\mathbf{cdr} \ x)$ is ϵ . Thus availability analysis involves an inward propagation of demand followed by an outward propagation of availability.

Given an expression e , the set of access paths σ describing the demand on the result of e and the availability environment \mathcal{A} at the program point before e , we compute the availability of e and the availability environment after e using the transfer function \mathcal{A} . This is described in Fig. 9. Availability analysis is an all paths problem. We get constraints involving intersection operation for sets describing availability. As intersection operation can not be mapped directly to CFGs, we need to get an approximate (but safe) solution. This is achieved by an intraprocedural analysis in which we neither propagate the demand from function application to its arguments, nor propagate the availability of arguments to the function application (29). A straightforward unfolding of \mathcal{A} will give us the availability environment at different program points.

5.2 Inward Propagation of Demand (\mathcal{AP})

If σ describes the set of paths specifying the demand on the result of evaluating the primitive application $(P \ e_1 \dots e_n)$ then $\mathcal{AP}_P^i(\sigma)$ gives the set of paths representing the demand on e_i .

$$\begin{aligned} \mathcal{AP}_{\mathbf{car}}^1(\sigma) &= \{\epsilon\} \cup \{\mathbf{0}\} \cdot \sigma & \mathcal{AP}_{\mathbf{cdr}}^1(\sigma) &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \sigma \\ \mathcal{AP}_{\mathbf{cons}}^1(\sigma) &= \{\mathbf{0}\} \cdot \sigma & \mathcal{AP}_{\mathbf{cons}}^2(\sigma) &= \{\mathbf{1}\} \cdot \sigma \\ \mathcal{AP}_{\mathbf{null?}}^1(\sigma) &= \emptyset, \quad \mathcal{AP}_{\mathbf{pair?}}^1(\sigma) = \emptyset, & \mathcal{AP}_{+}^1(\sigma) &= \emptyset, \quad \mathcal{AP}_{+}^2(\sigma) = \emptyset \end{aligned} \quad (30)$$

5.3 Outward Propagation of Availability (\mathcal{ABP})

If σ describes the availability of i^{th} argument of $(P \ e_1 \dots e_n)$, then $\mathcal{ABP}_P^i(\sigma)$ gives the availability of $(P \ e_1 \dots e_n)$. For the primitives in our language:

$$\begin{aligned} \mathcal{ABP}_{\mathbf{car}}^1(\sigma) &= \{\bar{\mathbf{0}}\} \cdot \sigma & \mathcal{ABP}_{\mathbf{cdr}}^1(\sigma) &= \{\bar{\mathbf{1}}\} \cdot \sigma \\ \mathcal{ABP}_{\mathbf{cons}}^1(\sigma) &= \{\epsilon\} \cup \{\mathbf{0}\} \cdot \sigma & \mathcal{ABP}_{\mathbf{cons}}^2(\sigma) &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \sigma \\ \mathcal{ABP}_{\mathbf{null?}}^1(\sigma) &= \emptyset, \quad \mathcal{ABP}_{\mathbf{pair?}}^1(\sigma) = \emptyset, & \mathcal{ABP}_{+}^1(\sigma) &= \emptyset, \quad \mathcal{ABP}_{+}^2(\sigma) = \emptyset \end{aligned} \quad (31)$$

$$\mathcal{A}\mathcal{E}(\kappa, \sigma, \mathcal{A}) = (\{\epsilon\}, \mathcal{A}) \quad (24)$$

$$\mathcal{A}\mathcal{E}(v, \sigma, \mathcal{A}) = (\mathbf{LET} \ \mathcal{A}' \leftarrow \text{update}(\mathcal{A}, v, \mathcal{A}(v) \cup \sigma) \ \mathbf{IN} \ (\mathcal{A}'(v), \mathcal{A}')) \quad (25)$$

$$\begin{aligned} \mathcal{A}\mathcal{E}((\mathbf{if} \ e_1 \ e_2 \ e_3), \sigma, \mathcal{A}) = & (\mathbf{LET} \ (\sigma_1, \mathcal{A}_1) \leftarrow \mathcal{A}\mathcal{E}(e_1, \{\epsilon\}, \mathcal{A}) \ \mathbf{IN} \\ & (\mathbf{LET} \ (\sigma_2, \mathcal{A}_2) \leftarrow \mathcal{A}\mathcal{E}(e_2, \sigma, \mathcal{A}_1) \ \mathbf{IN} \\ & (\mathbf{LET} \ (\sigma_3, \mathcal{A}_3) \leftarrow \mathcal{A}\mathcal{E}(e_3, \sigma, \mathcal{A}_1) \ \mathbf{IN} \\ & (\sigma \cup (\sigma_2 \cap \sigma_3), \mathcal{A}')))) \end{aligned} \quad (26)$$

$$\text{where } \mathcal{A}'(v) = \mathcal{A}_2(v) \cap \mathcal{A}_3(v)$$

$$\mathcal{A}\mathcal{E}((\mathbf{let} \ v_1 \leftarrow e_1 \ \mathbf{in} \ e_2), \sigma, \mathcal{A}) = (\mathbf{LET} \ (\sigma', \mathcal{A}') \leftarrow \mathcal{A}\mathcal{E}(e_1, \emptyset, \mathcal{A}) \ \mathbf{IN} \ \mathcal{A}\mathcal{E}(e_2, \sigma, \text{update}(\mathcal{A}', v_1, \sigma'))) \quad (27)$$

$$\mathcal{A}\mathcal{E}((P \ e_1 \ \dots \ e_n), \sigma, \mathcal{A}) = (\mathbf{LET} \ (\sigma_1, \mathcal{A}_1) \leftarrow \mathcal{A}\mathcal{E}(e_1, \mathcal{AP}_P^1(\sigma), \mathcal{A}) \ \mathbf{IN} \ (\mathbf{LET} \ (\sigma_2, \mathcal{A}_2) \leftarrow \mathcal{A}\mathcal{E}(e_2, \mathcal{AP}_P^2(\sigma), \mathcal{A}_1) \ \mathbf{IN} \ \dots \ (\mathbf{LET} \ (\sigma_n, \mathcal{A}_n) \leftarrow \mathcal{A}\mathcal{E}(e_n, \mathcal{AP}_P^n(\sigma), \mathcal{A}_{n-1}) \ \mathbf{IN} \ (\sigma \cup \bigcup_{1 \leq i \leq n} \mathcal{AP}_P^i(\sigma_i), \mathcal{A}_n)) \dots)) \quad (28)$$

$$\begin{aligned} \mathcal{A}\mathcal{E}((f \ e_1 \ \dots \ e_n), \sigma, \mathcal{A}) = & (\mathbf{LET} \ (\sigma_1, \mathcal{A}_1) \leftarrow \mathcal{A}\mathcal{E}(e_1, \emptyset, \mathcal{A}) \ \mathbf{IN} \\ f \text{ is a user defined function} \quad & (\mathbf{LET} \ (\sigma_2, \mathcal{A}_2) \leftarrow \mathcal{A}\mathcal{E}(e_2, \emptyset, \mathcal{A}_1) \ \mathbf{IN} \ \dots \ (\mathbf{LET} \ (\sigma_n, \mathcal{A}_n) \leftarrow \mathcal{A}\mathcal{E}(e_n, \emptyset, \mathcal{A}_{n-1}) \ \mathbf{IN} \ (\sigma, \mathcal{A}_n)) \dots)) \end{aligned} \quad (29)$$

Fig. 9. Computing $\mathcal{A}\mathcal{E}$

6 Null Insertion

We need to consider the following issues for null insertion:

- *Safety*: No live edge should be nullified. Further, the expression used to nullify an edge should not dereference a null reference.
- *Profitability*: An edge should be nullified as early as possible. Multiple nullification of same edge, through the same expression or through its aliases, should be avoided.

Safety, can be achieved by the following:

1. The proper prefixes of the access path used for nullification should be available. Thus, the candidate access paths at a given program point can be obtained by extending available access paths with a **0** and a **1**. Additionally, all root variables are also candidates for null insertion. The liveness of only these paths need to be checked for null insertion. Thus,

$$\mathbf{Candidates}(\pi) = \bigcup_{v \in \mathbf{SVars}(\pi)} v.(\{\epsilon\} \cup \{\alpha \mathbf{0}, \alpha \mathbf{1} \mid \alpha \in \mathcal{A}_\pi(v)\})$$

2. To make sure that the link described by a candidate path $v.\alpha$ is not live, we have to compute link aliases of $v.\alpha$ and ensure that none of them is live at π .

Our analyses annotate liveness, sharing and availability environments at the program points before the expressions. Therefore, we nullify dead links at these program points only. The analyses can easily be extended to compute the environments at the program points after the expressions, so that links can be nullified at these points.

To address the profitability issue, we visit the program points in the order of execution (reverse depth-first order of the expression tree) to nullify links. We mark the access paths which are already used for nullification, and do not nullify them again. However, redundant null insertions are still possible because the same link may be nullified more than once through aliased access paths. In general it is not possible to eliminate redundant null insertions. However, we can reduce them by computing *must-aliases* that hold on all paths, and marking all must-link-aliases of the access path used for nullification.

A given nullifiable access path can be translated into equivalent expression for nullification of the link it represents. We need three primitives in our meta language to achieve the effect of nullification. These are: **SET!** to nullify root variable, **SET-CAR!** to nullify **car** references, and **SET-CDR!** to nullify **cdr** references. The expression for nullification from access path is obtained using the function `Nullify` which is inserted at appropriate program points:

$$\begin{aligned} \text{Nullify}(v.\alpha) &= \begin{cases} (\mathbf{SET!} \ v \ \mathbf{nil}) & \alpha = \epsilon \\ \text{LinkNullify}(\alpha, v) & \alpha \neq \epsilon \end{cases} \\ \text{LinkNullify}(1\alpha, e) &= \begin{cases} (\mathbf{SET-CDR!} \ e \ \mathbf{nil}) & \alpha = \epsilon \\ \text{LinkNullify}(\alpha, (\mathbf{cdr} \ e)) & \alpha \neq \epsilon \end{cases} \\ \text{LinkNullify}(0\alpha, e) &= \begin{cases} (\mathbf{SET-CAR!} \ e \ \mathbf{nil}) & \alpha = \epsilon \\ \text{LinkNullify}(\alpha, (\mathbf{car} \ e)) & \alpha \neq \epsilon \end{cases} \end{aligned}$$

7 Related Work

Existing literature regarding improving memory usage can be categorized as follows:

Compile time reuse. The method by Barth [10] detects memory cells with zero reference count and reallocates them for further use in the program. Jones and Le Metayer [11] describe a sharing analysis based garbage collection for reusing of cells which collects a cell provided expressions using it do not need it for their evaluation.

Explicit reclamation. Shaham et. al. [12] use an automaton called *heap safety automaton* to model safety of inserting a free statement at a given program point. The analysis is based on shape analysis [13] and is very precise. However it is very inefficient. *Free-Me* [14] combines a lightweight pointer analysis with liveness information that detects when short-lived objects die and insert statements to free such objects. The analysis is simpler and cheaper as the scope is limited. The analysis described by Inoue et. al. [15] detects the scope (function) out of which a cell becomes unreachable, and explicitly claims the cell whenever the execution goes out of that scope. Like our method, the result of their analysis is also represented using CFGs. The main difference between their work and ours

is that we detect and nullify dead links at any point of the program, while they detect and collect objects that are unreachable at function boundaries.

Making dead objects unreachable. The most popular approach to make dead objects unreachable is to identify live variables and reduce the root set to only these variables [16]. The drawback of this approach is that all heap objects reachable from the live root variables are considered live, even if some of them remain unused. *Escape analysis* [17, 18] based approaches discover objects escaping a procedure, i.e. objects whose lifetimes outlive the procedure that created them. All non-escaping objects are allocated on stack, whereby they become unreachable whenever the creating procedure exits. *Region* based garbage collection [19] uses *region inference* [20] to identify regions that are allocated storage for objects. Memory blocks are always allocated in a particular region and are deallocated at the end of that region’s lifetime. Escape analysis and region inference detect garbage only at the boundaries of certain predefined areas of the program. In our previous work [21], we have used bounded abstractions of access paths called *access graphs* to describe the liveness of memory links in imperative programs and have used this information to nullify dead links. This paper is completion of our earlier work [6], where we used liveness to introduce the ideas presented in this paper.

8 Conclusions and Future Work

In this paper we have proposed a method to nullify links in heap memory to improve garbage collection. The method consists of a set of analyses to discover dead references at every program point followed by the actual insertion of null statements. We claim that the analyses are both scalable and precise—scalable because we obtain a context dependent summary of each function call, and precise because the summaries are used in a context- and flow-sensitive analysis of each function call. The method is very similar to the *functional method* of interprocedural analysis. However we have not found any published work which describes the functional method for non bit-vector problems.

This work can be extended in many directions. We can extend the language to include higher-order functions. The scope of the method can be extended to include dead-code elimination. If a reference to the value of `(cons e1 e2)` is never used, the expression need not be evaluated at all. Our method, in its present form, would first evaluate the expression and then nullify the reference to it. The safety of nullification has to be proven. Finally, the method has to be implemented to demonstrate its effectiveness.

References

1. Røjemo, N., Runciman, C.: Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In: ICFP, New York, NY, USA, ACM Press (1996) 34–41
2. Karkare, A., Sanyal, A., Khedker, U.: Effectiveness of garbage collection in mit/gnu scheme. <http://arxiv.org/abs/cs/0611093> (2006)

3. Shaham, R., Kolodner, E.K., Sagiv, M.: On the effectiveness of gc in java. In: MSP/ISMM. (2000) 12–17
4. Shaham, R., Kolodner, E.K., Sagiv, M.: Heap profiling for space-efficient java. In: PLDI. (2001) 104–113
5. Shaham, R., Kolodner, E.K., Sagiv, S.: Estimating the impact of heap liveness information on space consumption in java. In: MSP/ISMM. (2002) 171–182
6. Karkare, A., Khedker, U., Sanyal, A.: Liveness of heap data for functional programs. In: Heap Analysis and Verification Workshop. (2007)
7. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL. (1999) 133–146
8. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. In: OOPSLA. (1999) 187–206
9. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: CC. (2007)
10. Barth, J.M.: Shifting garbage collection overhead to compile time. *Commun. ACM* **20**(7) (1977) 513–518
11. Jones, S.B., Metayer, D.L.: Compile-time garbage collection by sharing analysis. In: FPCA, New York, NY, USA, ACM Press (1989) 54–74
12. Shaham, R., Yahav, E., Kolodner, E.K., Sagiv, S.: Establishing local temporal heap safety properties with applications to compile-time memory management. In: SAS. (2003) 483–503
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. *ACM TOPLAS* **24**(3) (2002) 217–298
14. Guyer, S.Z., McKinley, K.S., Frampton, D.: Free-me: a static analysis for automatic individual object reclamation. In: PLDI. (2006) 364–375
15. Inoue, K., Seki, H., Yagi, H.: Analysis of functional programs to detect run-time garbage cells. *ACM TOPLAS* **10**(4) (1988) 555–578
16. Agesen, O., Detlefs, D., Moss, J.E.: Garbage collection and local variable type-precision and liveness in Java virtual machines. In: PLDI. (1998) 269–279
17. Blanchet, B.: Escape analysis for JavaTM: Theory and practice. *ACM TOPLAS* **25**(6) (2003) 713–775
18. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: OOPSLA. (1999) 1–19
19. Hallenberg, N., Elsmann, M., Tofte, M.: Combining region inference and garbage collection. In: PLDI. (2002) 141–152
20. Tofte, M., Birkedal, L.: A region inference algorithm. *ACM TOPLAS* **20**(4) (1998) 724–767
21. Khedker, U., Sanyal, A., Karkare, A.: Heap reference analysis using access graphs. Submitted to ACM TOPLAS, copy available at <http://arxiv.org/abs/cs.PL/0608104> (2006)
22. Hopcroft, J.E., Ullman, J.D.: Introduction To Automata Theory, Languages, And Computation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)
23. Mohri, M., Nederhof, M.J.: Regular approximation of context-free grammars through transformation. In: Junqua, J.C., van Noord, G., eds.: Robustness in Language and Speech Technology. Kluwer Academic Publishers, Dordrecht (2000) 251–261

A Solving Liveness Equations

In general, the equations defining the functions \mathcal{LF} will be recursive. To solve such equations, we start by guessing that the solution will be of the form:

$$\mathcal{LF}_f^i(\sigma) = \mathcal{I}_f^i \cup \mathcal{D}_f^i \cdot \sigma,$$

where \mathcal{I}_f^i and \mathcal{D}_f^i are sets of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$. The intuition behind this form of solution is as follows: The function f can use its argument locally and/or copy a part of it to the return value being computed. \mathcal{I}_f^i is the set of live paths of i^{th} argument due to local use in f . \mathcal{D}_f^i is a sort of selector that selects the live paths corresponding to the i^{th} argument of f from σ , the liveness paths of the return value.

If we substitute the guessed form of \mathcal{LF}_f^i in the equations describing it and equate the terms containing σ and the terms without σ , we get the equations for \mathcal{I}_f^i and \mathcal{D}_f^i . This is illustrated in the following example.

Example 4. Consider the equation for $\mathcal{LF}_{\text{append}}^1(\sigma)$ from Example 1:

$$\mathcal{LF}_{\text{append}}^1(\sigma) = \{\epsilon\} \cup \{\mathbf{0}\bar{\mathbf{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{LF}_{\text{append}}^1(\{\bar{\mathbf{1}}\} \cdot \sigma)$$

Decomposing both sides of the equation, and rearranging gives:

$$\begin{aligned} \mathcal{I}_{\text{append}}^1 \cup \mathcal{D}_{\text{append}}^1 \cdot \sigma &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}_{\text{append}}^1 \\ &\quad \cup \{\mathbf{0}\bar{\mathbf{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{D}_{\text{append}}^1 \cdot \{\bar{\mathbf{1}}\} \cdot \sigma \end{aligned}$$

Separating the parts that are σ dependent and the parts that are σ independent, and equating them separately, we get:

$$\begin{aligned} \mathcal{I}_{\text{append}}^1 &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}_{\text{append}}^1 \\ \mathcal{D}_{\text{append}}^1 \cdot \sigma &= \{\mathbf{0}\bar{\mathbf{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{D}_{\text{append}}^1 \cdot \{\bar{\mathbf{1}}\} \sigma \\ &= (\{\mathbf{0}\bar{\mathbf{0}}\} \cup \{\mathbf{1}\} \cdot \mathcal{D}_{\text{append}}^1 \cdot \{\bar{\mathbf{1}}\}) \cdot \sigma \end{aligned}$$

As the equations hold for any general σ , we simplify them to:

$$\begin{aligned} \mathcal{I}_{\text{append}}^1 &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}_{\text{append}}^1 \\ \mathcal{D}_{\text{append}}^1 &= \{\mathbf{0}\bar{\mathbf{0}}\} \cup \{\mathbf{1}\} \cdot \mathcal{D}_{\text{append}}^1 \cdot \{\bar{\mathbf{1}}\} \end{aligned}$$

Similarly, from the equation describing $\mathcal{LF}_{\text{append}}^2(\sigma)$, we get:

$$\begin{aligned} \mathcal{I}_{\text{append}}^2 &= \mathcal{I}_{\text{append}}^2 \\ \mathcal{D}_{\text{append}}^2 &= \{\epsilon\} \cup \mathcal{D}_{\text{append}}^2 \cdot \{\bar{\mathbf{1}}\} \end{aligned}$$

These equations describe the transfer functions for `append`. \square

The values of \mathcal{I} and \mathcal{D} are sets of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$. We are interested in least solutions to the equations describing \mathcal{I} and \mathcal{D} . We use context free grammars (CFG) to describe these solutions. The set of terminal symbols of the CFG is $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$. Non-terminals and associated rules are constructed as illustrated in Examples 5 and 6.

Example 5. Consider the following constraint from Example 4:

$$\mathcal{I}_{\text{append}}^1 = \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}_{\text{append}}^1$$

We add non-terminal $\langle \mathcal{I}_{\text{append}}^1 \rangle$ and the productions with right hand sides directly derived from the constraints:

$$\langle \mathcal{I}_{\text{append}}^1 \rangle \rightarrow \epsilon \mid \mathbf{1} \langle \mathcal{I}_{\text{append}}^1 \rangle$$

The productions generated from other constraints of Example 4 are:

$$\langle \mathcal{D}_{\text{append}}^1 \rangle \rightarrow \mathbf{00} \mid \mathbf{1} \langle \mathcal{D}_{\text{append}}^1 \rangle \bar{\mathbf{1}}$$

$$\langle \mathcal{I}_{\text{append}}^2 \rangle \rightarrow \langle \mathcal{I}_{\text{append}}^2 \rangle$$

$$\langle \mathcal{D}_{\text{append}}^2 \rangle \rightarrow \epsilon \mid \langle \mathcal{D}_{\text{append}}^2 \rangle \bar{\mathbf{1}}$$

These productions describe the transfer functions of **append**. \square

The liveness environment at each program point can be represented as a CFG with a start symbol for every variable. To do so, the analysis starts with $\langle N_{\text{pgm}} \rangle$, the non-terminal describing the liveness of the result of the program, σ_{pgm} . The productions for $\langle N_{\text{pgm}} \rangle$ are:

$$\langle N_{\text{pgm}} \rangle \rightarrow \epsilon \mid \mathbf{0} \langle N_{\text{pgm}} \rangle \mid \mathbf{1} \langle N_{\text{pgm}} \rangle$$

Example 6. Let N_{π}^v denote the non-terminal corresponding to the liveness associated with a variable v at program point π . For the program of Fig. 1:

$$\begin{aligned} \langle N_{\pi_b}^w \rangle &\rightarrow \epsilon \mid \mathbf{1} \mid \mathbf{10} \mid \mathbf{100} \langle N_{\text{pgm}} \rangle \\ \langle N_{\pi_a}^z \rangle &\rightarrow \langle \mathcal{I}_{\text{append}}^2 \rangle \mid \langle \mathcal{D}_{\text{append}}^2 \rangle \mid \langle \mathcal{D}_{\text{append}}^2 \rangle \mathbf{1} \mid \langle \mathcal{D}_{\text{append}}^2 \rangle \mathbf{10} \\ &\quad \mid \langle \mathcal{D}_{\text{append}}^2 \rangle \mathbf{100} \langle N_{\text{pgm}} \rangle \\ \langle N_{\pi_a}^y \rangle &\rightarrow \langle \mathcal{I}_{\text{append}}^1 \rangle \mid \langle \mathcal{D}_{\text{append}}^1 \rangle \mid \langle \mathcal{D}_{\text{append}}^1 \rangle \mathbf{1} \mid \langle \mathcal{D}_{\text{append}}^1 \rangle \mathbf{10} \\ &\quad \mid \langle \mathcal{D}_{\text{append}}^1 \rangle \mathbf{100} \langle N_{\text{pgm}} \rangle \end{aligned}$$

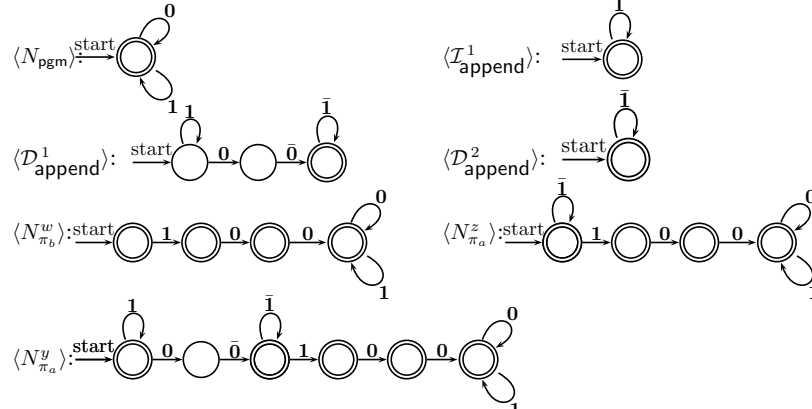
\square

It is possible that different paths, which are not in canonical form, may reduce to the same canonical path and hence encode the same information. We are interested in the information encoded by the paths, and therefore want to check memberships of canonical paths in CFGs. However, the paths described by the CFGs resulting out of our analysis are not in canonical form. It is not obvious how to check the membership of canonical paths directly in such CFGs. To solve this problem, we need equivalent CFGs such that if α belongs to an original CFG and $\alpha \xrightarrow{*} \beta$, where β is in canonical form, then β belongs to the corresponding new CFG. Directly converting the reduction rules (1) into productions and adding it to the grammar results in *unrestricted grammar* [22]. To simplify the problem, we approximate original CFGs by non-deterministic finite automata (NFAs) and convert them to equivalent NFAs which can be used to check the membership of canonical paths.

A.1 Approximating CFGs using NFAs

The conversion of a CFG G to an approximate NFA \mathbf{N} should be safe in that the language accepted by \mathbf{N} should be a superset of the language accepted by G . We use the algorithm described by Mohri and Nederhof [23]. The algorithm transforms a CFG to a restricted form called *strongly regular* CFG which can be converted easily to a finite automaton.

Example 7. We show the approximate NFAs for each of the non-terminals in Example 5 and Example 6.

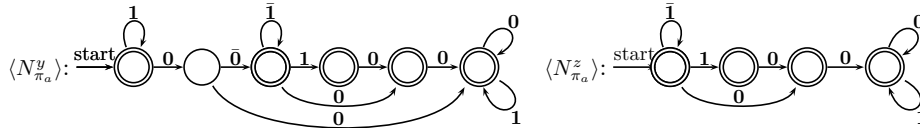


Note that there is no automaton for $\langle \mathcal{I}_{\text{append}}^2 \rangle$. This is because the least solution of the equation $\langle \mathcal{I}_{\text{append}}^2 \rangle \rightarrow \langle \mathcal{I}_{\text{append}}^2 \rangle$ is \emptyset . Also, the language accepted by the automaton for $\mathcal{D}_{\text{append}}^1$ is approximate as it does not ensure that there is an equal number of $\mathbf{1}$ and $\bar{\mathbf{1}}$ in the strings generated by rules for $\langle \mathcal{D}_{\text{append}}^1 \rangle$. \square

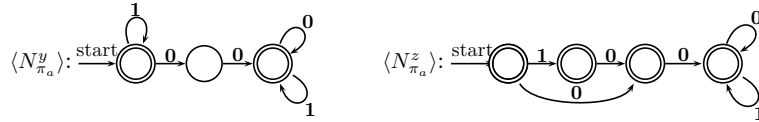
A.2 Conversion of NFAs to Accept Canonical Paths

Algorithm 1 converts an NFA with transitions on symbols $\bar{0}$ and $\bar{1}$ to an equivalent NFA without any transitions on these symbols. The algorithm repeatedly introduces ϵ edges to bypass a pair of consecutive edges labeled $\bar{0}\bar{0}$ or $\bar{1}\bar{1}$. The process is continued till a fixed point is reached. When the fixed point is reached, the resulting NFA contains the canonical paths corresponding to all the paths in the original NFA. The paths not in canonical form are deleted by removing edges labeled $\bar{0}$ and $\bar{1}$. Note that by our reduction rules if α is accepted by $\bar{\mathbf{N}}$ and $\alpha \xrightarrow{*} \perp$, then \perp should be accepted by \mathbf{N} . However, \mathbf{N} returned by our algorithm does not accept \perp . This is not a problem because the paths which are tested for membership against \mathbf{N} do not include \perp as well.

Example 8. We show the elimination of $\bar{0}$ and $\bar{1}$ for the automata for $\langle N_{\pi_a}^y \rangle$ and $\langle N_{\pi_a}^z \rangle$. The automaton for $\langle N_{\pi_b}^w \rangle$ remains unchanged as it does not contain transitions on $\bar{0}$ and $\bar{1}$. The automata at the termination of the loop in the algorithm are:



Eliminating the edges labeled $\bar{0}$ and $\bar{1}$, and removing the dead states gives:



Algorithm 1 Simplifying NFA

Input: An NFA \bar{N} with underlying alphabet $\{0, 1, \bar{0}, \bar{1}\}$

Output: An NFA \mathbf{N} with underlying alphabet $\{\mathbf{0}, \mathbf{1}\}$ accepting the equivalent set of paths

Steps:

$$i \leftarrow 0$$
$$N_0 \leftarrow \text{Equivalent NFA of } \overline{N} \text{ without } \epsilon\text{-moves [22]}$$

```
repeat
```

$$\mathbf{N}'_{i+1} \leftarrow \mathbf{N}_i$$

for all states q in \mathbf{N}_i such that q has an incoming edge from q' with label $\bar{\mathbf{0}}$ and outgoing edge to q'' with label $\mathbf{0}$ **do**

add an edge in \mathbf{N}'_{i+1} from q' to q'' with label ϵ . {bypass $\bar{00}$ using ϵ }

end for

for all states q in N_i such that q has an incoming edge from q' with label $\bar{1}$ and outgoing edge to q'' with label 1 **do**

add an edge in \mathbf{N}'_{i+1} from q' to q'' with label ϵ . {bypass $\bar{\mathbf{11}}$ using ϵ }

end for

$$\mathbf{N}_{i+1} \leftarrow \text{Equivalent NFA of } \mathbf{N}'_{i+1} \text{ without } \epsilon\text{-moves}$$
$$i \leftarrow i + 1$$

until ($\mathbf{N}_i = \mathbf{N}_{i-1}$)

$$\mathbf{N} \leftarrow \mathbf{N}_i$$

delete all edges with label $\bar{0}$ or $\bar{1}$ in N .

The language accepted by these automata represent the live access paths corresponding to y and z at π_a . \square

We now give the proofs of the termination and correctness of our algorithm.

Termination Termination of the algorithm follows from the fact that every iteration of **do-while** loop adds new edges to the NFA, while old edges are not deleted. Since no new states are added to NFA, only a fixed number of edges can be added before we reach a fix point.

Correctness The sequence of obtaining \mathbf{N} from $\overline{\mathbf{N}}$ can be viewed as follows, with \mathbf{N}_m denoting the NFA at the termination of while loop:

[illegible]

Then, the languages accepted by these NFAs have the following relation:

$$L(\overline{\mathbf{N}}) = L(\mathbf{N}_0) \subseteq \cdots \subseteq L(\mathbf{N}'_i) = L(\mathbf{N}_i) \subseteq \cdots = L(\mathbf{N}_m)$$

$$L(\mathbf{N}) \subseteq L(\mathbf{N}_m)$$

We first prove that the addition of ϵ -edges in the while loop does not add any new information, i.e. any path accepted by the NFA after the addition of ϵ -edges is a reduced version of some path existing in the NFA before the addition of ϵ -edges.

Lemma 1. *for $i > 0$, if $\alpha \in L(\mathbf{N}_i)$ then there exists $\alpha' \in L(\mathbf{N}_{i-1})$ such that $\alpha' \xrightarrow{*} \alpha$.*

Proof. As $L(\mathbf{N}_i) = L(\mathbf{N}'_i)$, we have $\alpha \in L(\mathbf{N}'_i)$. Only difference between \mathbf{N}'_i and \mathbf{N}_{i-1} is that \mathbf{N}'_i contains some extra ϵ -edges. Thus, any ϵ -edge free path in \mathbf{N}'_i is also in \mathbf{N}_{i-1} . Consider a path p in \mathbf{N}'_i that accepts α . Assume the number of ϵ edges in p is k . The proof is by induction on k .

(*BASE*) $k = 0$, i.e. p does not contains any ϵ -edge: As the path p is ϵ -edge free, it must be present in \mathbf{N}_{i-1} . Thus, \mathbf{N}_{i-1} also accepts α . $\alpha \xrightarrow{*} \alpha$.

(*HYPOTHESIS*) For any $\alpha \in L(\mathbf{N}_i)$ with accepting path p having less than k ϵ -edges there exists $\alpha' \in L(\mathbf{N}_{i-1})$ such that $\alpha' \xrightarrow{*} \alpha$.

(*INDUCTION*) p contains k ϵ -edges e_1, \dots, e_k : Assume e_1 connects states q' and q'' in \mathbf{N}'_i . By construction, there exists a state q in \mathbf{N}'_i such that there is an edge e'_1 from q' to q with label $\bar{0}(\bar{1})$ and an edge e''_1 from q to q'' with label $0(1)$ in \mathbf{N}'_i . Replace e_1 by $e'_1 e''_1$ in p to get a new path p'' in \mathbf{N}'_i . Let α'' be the path accepted by p'' . Clearly, $\alpha'' \xrightarrow{1} \alpha$. Since p'' has $k - 1$ ϵ -edges, α'' is accepted by \mathbf{N}'_i along a path (p'') that has less than k ϵ -edges. By induction hypothesis, we have $\alpha' \in L(\mathbf{N}_{i-1})$ such that $\alpha' \xrightarrow{*} \alpha''$. This along with $\alpha'' \xrightarrow{1} \alpha$ gives $\alpha' \xrightarrow{*} \alpha$.

Corollary 1. *for each $\alpha \in L(\mathbf{N}_m)$, there exists $\alpha' \in L(\bar{\mathbf{N}})$ such that $\alpha' \xrightarrow{*} \alpha$.*

Proof. The proof is by induction on m , and using Lemma 1.

The following lemma shows that the the language accepted by \mathbf{N}_m is closed with respect to reduction of paths.

Lemma 2. *For $\alpha \in L(\mathbf{N}_m)$, if $\alpha \xrightarrow{*} \alpha'$ and $\alpha' \neq \perp$, then $\alpha' \in L(\mathbf{N}_m)$.*

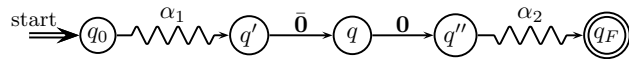
Proof. Assume $\alpha \xrightarrow{k} \alpha'$. The Proof is by induction on k , number of steps in reduction.

(*BASE*) case $k = 0$ is trivial as $\alpha \xrightarrow{0} \alpha$.

(*HYPOTHESIS*) Assume that for $\alpha \in L(\mathbf{N}_m)$, if $\alpha \xrightarrow{k-1} \alpha'$, then $\alpha' \in L(\mathbf{N}_m)$.

(*INDUCTION*) $\alpha \in L(\mathbf{N}_m)$, $\alpha \xrightarrow{k} \alpha'$. There exists α'' such that: $\alpha \xrightarrow{k-1} \alpha'' \xrightarrow{1} \alpha'$. By induction hypothesis, we have $\alpha'' \in L(\mathbf{N}_m)$.

For $\alpha'' \xrightarrow{1} \alpha'$ to hold we must have $\alpha'' = \alpha_1 \bar{0} 0 \alpha_2$ and $\alpha' = \alpha_1 \alpha_2$, or $\alpha'' = \alpha_1 \bar{1} 1 \alpha_2$ and $\alpha' = \alpha_1 \alpha_2$. Consider the case when $\alpha'' = \alpha_1 \bar{0} 0 \alpha_2$. Any path in \mathbf{N}_m accepting α'' must have the following structure (The states shown separately may not necessarily be different):



As \mathbf{N}_m is the fixed point NFA for the iteration process described in the algorithm, adding an ϵ -edge between states q' and q'' will not change the language accepted by \mathbf{N}_m . But, the path accepted after adding an ϵ -edge is $\alpha_1\alpha_2 = \alpha'$. Thus, $\alpha' \in L(\mathbf{N}_m)$. The case when $\alpha'' = \alpha_1\bar{1}\alpha_2$ is identical.

Corollary 2. *For $\alpha \in L(\bar{\mathbf{N}})$, if $\alpha \xrightarrow{*} \alpha'$ and $\alpha' \neq \perp$, then $\alpha' \in L(\mathbf{N}_m)$.*

Proof. $L(\bar{\mathbf{N}}) \subseteq L(\mathbf{N}_m) \Rightarrow \alpha \in L(\mathbf{N}_m)$. The proof follows from Lemma 2.

The following theorem asserts the equivalence of $\bar{\mathbf{N}}$ and \mathbf{N} with respect to the equivalence of paths, i.e. every path in $\bar{\mathbf{N}}$ has an equivalent canonical path in \mathbf{N} , and for every canonical path in \mathbf{N} , there exists an equivalent path in $\bar{\mathbf{N}}$.

Theorem 1. *Let $\bar{\mathbf{N}}$ be an NFA with underlying alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$. Let NFA \mathbf{N} be the NFA with underlying alphabet $\{\mathbf{0}, \mathbf{1}\}$ returned by the algorithm. Then,*

1. *if $\alpha \in L(\bar{\mathbf{N}})$, β is a canonical path such that $\alpha \xrightarrow{*} \beta$ and $\beta \neq \perp$, then $\beta \in L(\mathbf{N})$.*
2. *if $\beta \in L(\mathbf{N})$ then there exists a path $\alpha \in L(\bar{\mathbf{N}})$ such that $\alpha \xrightarrow{*} \beta$.*

Proof.

1. From Corollary 2:
 $\alpha \in L(\bar{\mathbf{N}}), \alpha \xrightarrow{*} \beta$ and $\beta \neq \perp \Rightarrow \beta \in L(\mathbf{N}_m)$. As β is in canonical form, the path accepting β in \mathbf{N}_m consists of edges labeled $\mathbf{0}$ and $\mathbf{1}$ only. The same path exists in \mathbf{N} . Thus \mathbf{N} also accepts $\beta \Rightarrow \beta \in L(\mathbf{N})$.
2. $L(\mathbf{N}) \subseteq L(\mathbf{N}_m) \Rightarrow \beta \in L(\mathbf{N}_m)$. Using Corollary 1, there exists $\alpha \in L(\bar{\mathbf{N}})$ such that $\alpha \xrightarrow{*} \beta$.